The Dynamic Function Coupling Metric and Its Use in Software Evolution

Árpád Beszédes, Tamás Gergely, Szabolcs Faragó, Tibor Gyimóthy and Ferenc Fischer
University of Szeged, Department of Software Engineering
Árpád tér 2., H-6720 Szeged, Hungary, +36 62 544145
{beszedes,gertom,faragosz,gyimi}@inf.u-szeged.hu
Fischer.Ferenc@stud.u-szeged.hu

Abstract

Many of the existing techniques for impact set computation in change propagation and regression testing are approximate for the sake of efficiency. A way to improve precision is to apply dynamic analyses instead of static ones. The state-of-the-art dynamic impact analysis method is simple and efficient, but overly conservative and hence imprecise. In this paper we introduce the measure of Dynamic Function Coupling (DFC) *between two functions or methods, which we use to define a more precise way of computing impact sets on function level with a scalable rate of recall. The intuition behind our approach is that the 'closer' the execution of a function is to the execution of another function in some of the runs of the program, the more likely they are really dependent on each other. So, impact sets may be computed based on this kind of coupling. We provide experimental data to support the validity of the concept, which essentially show that the impact set of a function consisting of only strongly DFC-coupled functions has twice the precision compared to the conservative method.*

Keywords

Software maintenance and evolution, change impact analysis, regression testing, dynamic program analysis.

1 Introduction

Impact analysis [6] is a very important software engineering activity, especially in software processes with a high degree of incrementality, change propagation [15] and regression testing [17] being two of its main application fields. Different approaches exist for impact set computation, which is the main purpose of impact analysis. Both static and dynamic methods are being investigated, however, recently the focus of several researchers has been shifted towards the latter due to the overly conservative (hence imprecise) nature of traditional static approaches.

The current state-of-the-art method for producing dynamic impact sets on function level is based on the so-called Execute After sequences computed from execution traces with function entry and exit events [2]. Apiwattanapong *et al.* use a very simple approach that essentially states the following: based on a set of executions, a specific function f will potentially have an impact on all those methods that are executed sometime *after it* in any of the executions, meaning that any function g executed after f will become part of f's impact set. Note, that this is very simple to compute since one only has to look at the first occurrence of f in the trace and consider all other functions located after this position in the trace. This approach is safe—meaning that no dependence is missed—, but very imprecise too (in the following we will refer to this method as the *conservative method*). In fact, based purely on the sequence of function calls and returns, it seems to be impossible to provide a more precise, yet still safe method.

In this paper, we further develop the notion of Execute After sequences by tackling two problems with it. First, for change propagation, we advocate that both directions of potential dependencies need to be taken into account (the *impacting* and the *impacted* program elements) and incorporated into the impact set [15]. However, for regression testing the forward computation impact is sufficient, meaning that the modification program point and the impacted elements need to be considered for determining the test cases for retesting (will use the *computation impact* terminology here). Our second enhancement is that we give a method for computing more precise impact sets with the trade-off of loosing the safety of the approach. The basic idea for refining Execute After relations between the functions is based on the intuition that the 'closer' the execution of a function f is to the execution of function g in some of the runs of the program, the more likely they are dependent on each other.

But, what a 'close' execution should mean? Simply limiting the impact set of f to a simple surrounding with radius of some fixed value d of each of f's occurrences in the traces is invalid because of other possible intervening func-

tion calls made by f. Defining the impact sets based on the dynamic call graph such that the impact set of function f contains the reachable functions g (in both directions) up to a fixed distance d, is a much better approximation (we call this a *call-indirection* relation between functions). However, many further dependencies may be missed using this approach. Namely, if a function f is called by a function h and later g is called as well by h, g may be dependent on f if f sets a data that g reads (we will refer to this kind of dependency as the *sequence-indirection*).

Thus, our approach is the following. We define the measure of *Dynamic Function Coupling* (DFC) between two functions as the minimal level of indirection between all possible occurrences of the two functions in the traces (the different executions of the program contribute jointly to this measure, so representative test cases induce more accurate DFC values). Informally, the level of indirection is the 'closeness' of the two functions taking into account the number of other intervening functions at places where call- or sequence-indirection takes place.

Once we have the DFC metric for every pair of functions, we may compute the impact set of a function f by taking those functions that have a DFC of at most some fixed cut-off value d. Based on this heuristic, in this paper we give a method for computing the impact sets using a fixed cut-off value, and promote the use of such sets for mentioned activities instead of (1) the imprecise Execute After relations, (2) the precise but expensive dynamic slices, and (3) imprecise and/or unsafe static dependency sets.

The indirection level d serves as a parameter to our impact set computing algorithm, which opens the possibility to balance between precision and recall. For example, as a special case the algorithm is able to compute the original Execute After relation with an infinite d (safe but imprecise), and on the other end, only directly coupled functions can be retrieved as well (precise and small but unsafe).

We investigate the validity of the approach and assess its efficiency in terms of (1) precision/recall measurement with respect to precisely computed dynamic dependencies among functions, and in terms of (2) the amount of reduction in the impact set sizes compared to the conservative method. For this validation we rely on fine-grained dynamic control and data dependencies computed globally for many slicing criteria [4] using our dynamic slicing tool for Java [19]. So we consider that there is an *actual coupling* between the functions if there is at least one fine-grained dynamic dependency between them.

We define the following research questions to be experimentally answered in this article.

RQ1 Is it true that a small DFC value between two functions (close indirection) indicates a more probable actual coupling between them?

RQ2 To what extent do call-indirections alone and together with the sequence-indirections reflect actual coupling?

RQ3 What is the threshold value of parameter d that produces good recall with respect to the fine-grained dynamic dependencies, and what is the precision of the method with that parameter?

RQ4 If the application of the method requires smaller sets and better precision, what d values would produce such sets, and what is the recall in that case?

RQ5 How much gain can we achieve using this method compared to the conservative approach in terms of the size of the impact sets?

In the next section we overview related work. In Section 3 we define the DFC measure and the method for computing it and the impact sets themselves. Section 4 deals with the ways this method can be applied to different tasks in software evolution. In Section 5 we describe our experiments with test programs and discuss the results. In Section 6 a discussion of our findings with answers to the research questions is given, while the final section is dedicated to the conclusions and future work.

2 Related work

Many common methods for change impact analysis are static, e. g. [15, 16]. In their work, Rajlich and Gosavi define a lifecycle model that heavily utilizes static change impact analysis, while Buckner *et al.* provide an experimental tool supporting this paradigm [10].

Our algorithm for computing dynamic impact sets on function level is motivated by and is the generalization of the work of Apiwattanapong *et al.* [2]. The function-level dynamic impact analysis technique by Orso *et al.* [14] combines approximate static forward slices with coverage information to obtain the dynamic impact sets. The dynamic computation part is simple and efficient, however it misses information about the ordering of function calls. The static analysis part of the algorithm may be expensive for large programs, furthermore it is imprecise too, since it essentially employs a flow-graph reachability approach. Law and Rothermel presented another function-level dynamic impact algorithm [12], which is based on trace traversal. Although it is more precise, it incurs much higher overhead in time and space due to being dependent on the size of the trace (the authors provide, though, a way to compress the traces efficiently).

Our work partially covers the method by Breech *et al.* for dynamic impact analysis [7]. The authors present an online algorithm for computing dynamic impact sets based on

dynamic call graphs, which directly corresponds to the reduced version of our algorithm involving only forward and backward call-indirections (EA_{call}, EA_{ret}).

A very common application of impact analysis is in regression testing, more specifically as a means for selective retesting (Rothermel and Harrold give an excellent comparative overview of regression test selection techniques [17]). Most of the existing methods in this field are based on static program analysis [6]. For example, Rothermel and Harrold use control flow graphs for safe test selection under certain conditions [18]. Other notable static approaches are by Binkley [5] and Gupta *et al.* [11]. A certain class of practically usable methods are less precise but very efficient, such as the application of testing firewalls [20, 21].

Agrawal *et al.* use different dynamic analysis techniques combined with static analyses and give several incremental algorithms for regression testing, including the application of relevant slices for this purpose [1]. These and similar dynamic analysis-based approaches suffer from the inherent problem of very expensive computation due to the large amount of dynamic data to be processed. More lightweight methods can produce less precise results, however they can be applied in real life scenarios. For example, Orso *et al.* present the way of applying their function-level impact analysis method for regression testing of real applications and field data [14].

Different kinds of static coupling metrics have been proposed in the literature [8] (their usefulness specifically for impact analysis has been verified by Briand *et al.* [9]). Dynamic program metrics is a much less elaborated research field. Arisholm *et al.* evaluated a number of dynamic coupling metrics in object oriented software and found that some of them complement static coupling measures [3]. Mitchell and Power investigated different dynamic coupling between objects metrics [13]. The authors measured the correlation between static and dynamic measures, and found that they are quite independent from each other, suggesting that static metrics alone are not so good predictors of runtime behavior of objects, only combined with coverage information.

3 Dynamic Function Coupling

Apiwattanapong *et al.* [2] defined the EA (Execute After) relation between two methods f and g, given a program and an execution of it, in the following way (in order to be consistent with our notation we transposed the relations, and we also omitted the incorporation of more than one execution). $(f, g) \in EA$ iff f calls g (directly or transitively), or f returns into g (directly or transitively), or f returns into a method h (directly or transitively), and method h later calls g (directly or transitively).

We will follow this threefold categorization of the relation between executed functions as it covers all possibilities of their sequential execution (and possible interaction thereof), and provide some refinements to it as described in the following.

We rely on *function entry* and *function exit* events in the trace. To simplify the formal definitions below, we will use the concept of *dynamic call trees*, which can be constructed based on these events very easily using a traditional call-stack approach. Note however, that actually building this representation is not required by the implementation of our method, it is used for demonstration purposes only. For a program P and its execution with the trace T we define the dynamic call tree[1] as a rooted tree G with the ordering of the neighbors at each vertex. The vertices of G represent the instances of functions in T, while the root is the *main* function of the program (the first function executed). The children of a vertex v represent the functions v directly called at that point taking into consideration the order of the invocations. We will also use the term *f-to-g call chain* in G, which is a path from vertex v to vertex u, these being instances of functions f and g, respectively, for which the following holds: the path from the root to v is the prefix of the path from the root to u.

Following the intuition outlined above, let us extend the definition of Execute After with the measure of indirection level d. We give the definition by the separation of the three cases when two functions can be executed after each other, and define the following relations:

$$
\begin{aligned}
(f,g) \in EA^{(d)}_{call} &\overset{\text{def}}{\iff} \exists\, f\text{-to-}g \text{ call chain of length } d,\\
(f,g) \in EA^{(d)}_{ret} &\overset{\text{def}}{\iff} \exists\, g\text{-to-}f \text{ call chain of length } d,\\
(f,g) \in EA^{(d)}_{seq} &\overset{\text{def}}{\iff} \exists \text{ function } h \text{ with:}\\
&\quad h\text{-to-}f \text{ call chain of length } d_r \text{ and}\\
&\quad h\text{-to-}g \text{ call chain of length } d_c,\\
&\quad \text{having a common instance vertex for } h,\\
&\quad \text{where } f \text{ is called first, and}\\
&\quad d = d_r + d_c - 1.
\end{aligned}
$$

Based on these, $EA^{(d)}$ will be the combined *Execute After* relation, which permits the maximal (cut-off) indirection level of d, formally defined as follows:

$$
\begin{aligned}
(f,g) \in EA^{(d)} &\overset{\text{def}}{\iff}\\
&\exists\, d' \le d : (f,g) \in EA^{(d')}_{call} \cup EA^{(d')}_{ret} \cup EA^{(d')}_{seq}.
\end{aligned}
$$

Following our view on the symmetry of the coupling of

[1] In this paper we concentrate on the analysis of single threaded programs and executions with "regular" function-entry and -exit sequences, having the following properties: every function call induces a pair of entry-exit events, and these events follow a clean recursive structure.

two functions based on their interaction, we define the *Execute Before* relation ($EB^{(d)}$) very simply by reversing the roles of the two functions, for any d:

$$(f,g) \in EB^{(d)} \stackrel{\text{def}}{\iff} (g,f) \in EA^{(d)},$$

and by combining these two relations we define the *Execute Round* relation as well, as follows:

$$\forall d : ER^{(d)} = EB^{(d)} \cup EA^{(d)}.$$

Observe, that as special cases of our definitions, $EA^{(\infty)}$ corresponds to Apiwattanapong *et al.*'s definition of the Execute After relation, while $ER^{(\infty)}$ gives the complete graph with the covered functions. It is also easy to verify that the *Execute Round* relation will be symmetric, while the other two are not. Also note, that for computing ER only one of EA_{call} or EA_{ret} are needed since they are defined to be symmetric just like ER, however for the sake of completeness we use the definitions above.

Naturally, if a cut-off level d is sufficient for a pair of functions to be connected by *Execute Round*, all higher levels will be appropriate too. So, it is interesting to see what is the *lowest* of such levels for a pair of functions, and eventually, this will become the Dynamic Function Coupling measure that we informally defined earlier. Thus:

$$DFC(f,g) = \begin{cases} \min\{d \mid (f,g) \in ER^{(d)}\} & \text{if such } d \text{ exists,} \\ \infty & \text{otherwise.} \end{cases}$$

Observe that $DFC(f,g) = DFC(g,f)$ and $DFC(f,f) = 0$ will be true for any two functions f and g.[2]

The definitions above are given for one specific execution of a program, however they can be easily extended to multiple executions and thus providing the combined data for a complete set of test cases, where such is required like in regression testing. Namely, the EA, EB and ER relations for a set of executions can be obtained by computing the respective unions of the individual relations, while the DFC metric for any two functions will be the minimal such value from all of the executions. For the sake of simplicity, in the rest of the paper we will assume that the computed data are obtained from a fixed set of test cases, however more investigation is needed on the sensitivity of the approach to different attributes of the test cases.

Based on the above, we formally give our way of computing dynamic impact sets. For a program, a set of test cases and a fixed indirection cut-off value d, the dynamic impact set of a set of changed functions C is the following:

$$ImpactSet^{(d)}(C) = \{g \mid \exists f \in C : (f,g) \in ER^{(d)}\},$$

and similarly, the computation impact set is the following:

$$CImpactSet^{(d)}(C) = \{g \mid \exists f \in C : (f,g) \in EA^{(d)}\}.$$

[2]Here we do not follow the traditional convention that a larger value means stronger coupling.

3.1 Example

Let $T = \langle f_e, g_e, h_e, h_r, k_e, g_e, g_r, k_r, l_e, l_r, g_r, k_e, g_e, g_r, f_e, f_r, k_r, f_r \rangle$ be a trace (e: entry, r: return), which produces the dynamic call tree shown in Figure 1.

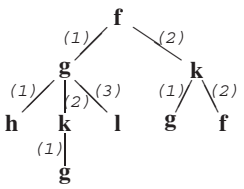


Figure 1. A dynamic call tree. The numbers on the edges correspond to the order of invocations.

The left-hand side part of the table that follows shows the minimal call-indirection level for each function pair (minimal d for which $(row, column) \in EA^{(d)}_{call}$), while the sequence-indirection levels are on the right-hand side:

d	f	g	h	k	l	f	g	h	k	l
f	0	1	2	1	2	∞	∞	∞	∞	∞
g	∞	0	1	1	1	1	2	∞	1	2
h	∞	∞	0	∞	∞	3	2	∞	1	1
k	1	1	∞	0	∞	3	1	∞	2	1
l	∞	∞	∞	∞	0	3	3	∞	2	∞

By transposing both matrices and selecting the minimal value from the four matrices in each cell, we get the DFC measures shown below:

DFC	f	g	h	k	l
f	0	1	2	1	2
g	1	0	1	1	1
h	2	1	0	1	1
k	1	1	1	0	1
l	2	1	1	1	0

In the following we give algorithms for computing Execute Round relations, however they can be easily modified to compute only Execute After relations.

3.2 Global algorithm for DFC

In this subsection we give the algorithm we used in our experiments. It computes globally the DFC metric for each function pair, and although it can be used to determine the ER relations and the impact sets, it is not practical due to its linear or quadratic complexity. A practical on-demand

algorithm for a fixed d value is presented in the next subsection.

The algorithm works on a trace (denoted by T) of length t, which is essentially a stream containing function entry and exit events. The maximal depth of the dynamic call tree corresponding to the trace will be denoted by m, while the number of functions covered will be n.

The algorithm shown in Figure 2 is a recursive one. For an actual subtree of the dynamic call tree rooted at function h it works as follows. For all subtrees of h it first computes minimal h-to-f call chain lengths recursively (line 8), and then the sequence-indirection levels while h being the "root" between any two f and g functions (lines 9–13). It then updates call-indirection levels with h (lines 16–19) and returns with updated call chain length information.

```
program  ComputeDFC(T)
input:   T : trace
output:  D[f,g] : DFC between all functions f and g
begin
1   init all elements in D with ∞
2   E := Read T
3   ComputeDistances(E.function)
4   Output D
end
procedure ComputeDistances(function h)
local:   P[f] : array of previous values
         N[f] : array of next values
begin
5   init all elements in P with ∞
6   E := Read T
7   while E is an ENTRY event
8      N := ComputeDistances(E.function)
9      forall f functions
10        forall g functions
11           D[f,g] := min(D[f,g], P[f] + N[g] − 1)
12           D[g,f] := D[f,g]
13     P := min(P, N)
14     E := Read T
15  P[h] := 0
16  forall f functions
17    D[h,f] := min(D[h,f], P[f])
18    D[f,h] := D[h,f]
19    P[f] := P[f] + 1
20  return P
end
```

Figure 2. Global DFC algorithm

Per trace element, this algorithm requires $O(n^2)$ time, while its memory requirement is $O(n \cdot m)$ not counting the DFC matrix itself.

We feel important to note that it is possible to construct a non-recursive algorithm for computing DFC values globally that has $O(n)$ time complexity per event. However, since it is more complicated to explain and due to space constraints we will not present this algorithm here.

3.3 On-demand algorithm for ER

The algorithm presented in the previous section is global meaning that it computes the relation between all function pairs. Based on it, it is trivial to determine the impact set of a set of changed functions. However, this is not so efficient in practice since we are generally interested in the impact of only a small number of functions.

In Figure 3 we present the algorithm that computes the set of impacted functions for a given changed function set using the ER relation with an indirection level d. Processing the trace, the algorithm maintains $n + 2$ stacks. The tops of $CALL$ and RET show the indirection levels from the last changed function according to EA_{call} and EA_{ret}, respectively, while the top of $SEQ[g]$ shows the indirection from g, which corresponds to EA_{seq}.

On an entry event, new values are pushed onto the stacks depending on whether the entered function is a changed one or not (lines 6–8). The resulting impact set is also updated according to the stack tops (lines 12–20). On return events the stacks are updated by popping them, and the new tops of RET and SEQ sets are updated using the popped value based on whether the returned function is a changed one or not (lines 23–28).

The time requirement of this algorithm is $O(n)$ per trace element, if appropriate data structures are used. The memory requirement is $O(n \cdot m)$.

Since our method for computing impact sets is often most successfully applied with a small indirection level d, it is useful to investigate another more specialized versions of it for fixed d values. Due to space constraints, we will not present the specialized algorithms, but we will note that because of some possibilities for simplifications in the data structures used (for example, SEQ does not need to be a vector), significantly better complexity requirements can be achieved. Namely, the algorithm with $d = 1$ has $O(m \cdot log(n))$ time- and $O(n \cdot m)$ space cost in the worst case, and the average requirements are even better.

However, in some cases even this complexity could be unacceptable. Then, a reduced version of the method incorporating only call-indirections can be used. We do not present this algorithm here, we just note that it can be implemented in $O(1)$ time with respect to each step of the trace.

4 Applications

Since the DFC metric is computed from a set of test cases, it represents the actual coupling level between func-

```
program  ComputeER(T, C, d)
input:   T : trace
         C : set of changed functions
         d : cut-off indirection level
output:  IMP : change impact set of C
data:    CALL, RET : stacks of values
         SEQ[] : vector of stacks with values
begin
1   IMP := C
2   init all stacks by pushing ∞ in them
3   while T is not empty
4     E := Read T
5     f := E.function
6     if E is an ENTRY event then
7       if f ∈ C then push(0, CALL))
8       else push(top(CALL) + 1, CALL))
9       push(RET, ∞)
10      forall g functions
11        push(SEQ[g], top(SEQ[g]) + 1)
12      if top(CALL) < d then
13        insert f into IMP
14      forall g ∈ C functions
15        if top(SEQ[g]) < d then
16          insert f into IMP
17      if f ∈ C then
18        forall g ∉ IMP functions
19          if top(SEQ[g]) < d then
20            insert g into IMP
21    else
22      pop(CALL)
23      u := min(pop(RET) + 1,pop(RET))
24      if f ∈ C then push(RET, 1)
25      else push(RET, u)
26      forall g functions
27        if f ∈ C then push(SEQ[g], 0)
28        else push(SEQ[g], u)
29      if top(RET) < d then
30        insert f into IMP
end
```

Figure 3. ER **algorithm**

tions with respect to that test suite. If the test suite represents the relevant use cases of the system, it actually may serve as a replacement to related static metrics [9]. Indeed, according to some studies, dynamic coupling metrics are better predictors of runtime behavior of objects than their static counterparts [13]. Thus, practically any application where coupling metrics are useful, DFC may be applied too. It could also be extended to class level using a suitable combination of all methods' DFC values of a class.

However, the most promising application fields of this metric are change impact analysis, regression testing and debugging. The possibility for parameterizing the computation algorithm with the cut-off value d enables its flexible application in these fields, namely, using infinite d for safe but imprecise sets, and smaller d values for smaller and more precise returned sets with accordingly worse recall.

For illustration, in this section we will outline a possible usage scenario in the mentioned fields. The benefit of using our method instead of previous ones will be elaborated based on the experimental results in the next section.

4.1 Change impact analysis

In change impact analysis [15], when a change is made to a part of the system, the other parts of the system that need to be investigated in order to propagate the change may be computed using $ImpactSet^{(d)}$ with a fixed d. A usable scenario could be the following: the initial impact sets are computed for all functions during a regular all-inclusive testing process (this can be done in parallel using our algorithm). This database is then used in subsequent activities when incremental changes are made to the system. After a change however, it will not necessarily reflect the actual impact sets anymore, so the database needs to be maintained. It may be used for a certain period of time as it is, if a change propagation method can tolerate a certain level of inconsistency, and after this period the database is completely regenerated. Alternatively, the entries in the database could be regularly updated corresponding to the changed functions, by extending the impact sets with the newly appearing dependencies (these can be computed by rerunning the affected tests, which is probably done anyway due to retesting after the change). However, the impact sets cannot be reduced with this method, so the corresponding parts of the database would need to be regenerated from time to time based on last modification times.

Some change propagation methods rely on high recall, in which case we can use a large d value (according to our experiments the smallest values where it reached 100% were around 8–15). On the other hand, other approaches benefit from better precision, in which case a close coupling ($d = 1$) should be chosen.

4.2 Regression testing

As far as regression testing is concerned, $CImpactSet^{(d)}$ can be used, again with a fixed d. With this kind of computation of impact sets, generally all traditional modification-based test selection strategies may be used. Namely, we may obtain a database of impact sets similarly to as outlined for change impact analysis above, and the test cases that need to be retested may be selected using these impact sets. Also, the same possibilities may used here for the maintenance of the database.

Regression testing may also benefit from the possibility to parameterize the algorithms with the d value. For example, testing firewalls are typically defined to involve only the closest dependents, in which case our impact sets with $d = 1$ can be a good alternative. Other regression testing approaches require the impact sets to be safer, in which case larger cut-offs may be chosen.

4.3 Debugging

Program debugging can also benefit from the coupling relation introduced in this article. Consider a scenario where a faulty value is observed at a specific program point, which is then marked with a breakpoint. It is obvious that debugging the program step-by-step starting at the beginning of the execution is not the optimal strategy. Inserting additional breakpoints in the code could help a lot, but the question is where to put them. For this problem a heuristic can be used in which, using the $EB^{(d)}$ relation, we can determine the set of functions that may have probable effect on the erroneous value, and put breakpoints at the exit points of these functions. The return values of these functions can then be checked, whose incorrectness may indicate a bug's occurrence. Varying the d value gives us the flexibility so that if a small value does not uncover the place of the bug, we still can use a higher value until the bug has been found. This method will probably produce more usable breakpoints and will help finding the bug faster than random breakpoint selection.

5 Experiments

In this section we experimentally evaluate the DFC metric and the impact sets computed based on it in terms of the sets' size and quality. The set sizes are important since they indicate the amount of reduction that can be achieved compared to the conservative method. The quality of the sets, on the other hand, is assessed through measuring the precision and recall rates with respect to precisely computed dynamic dependencies among functions.

5.1 Experimental tool setup

We performed our experiments on three medium size open source Java programs and a set of test cases for each of them. The programs were JSubtitles (15 classes, 460 lines), NanoXML (27 classes, 1156 lines) and java2html (55 classes, 2290 lines) with 95–100 test cases each.

For computing precision and recall, we applied fine-grained, instruction level dynamic dependencies computed with our Java dynamic slicer called Jadys [19]. This tool is able to compute dynamic slices on the lowest level of granularity and for all possible criteria globally. The so produced dynamic slices were lifted to method level and used as base information on actual couplings. To produce the relations introduced in this paper for different d values, we applied our global algorithm for DFC using matrices similar to the examples in Section 3. Taking advantage of the symmetries between the different kinds of indirection relations, only two matrices were actually computed: EA_{call} and EA_{seq}. The matrices of other relations were then computed using transposition and minimization.

Figure 4 shows our experimental toolchain. To compute the actual couplings the measured program is first executed on an instrumented version of the open source virtual machine JamVM, which generates an instruction level trace. Then the Jadys tool processes this trace and computes forward and backward statement-level dynamic slices. The J4J utility then propagates instruction level information to method level, thus producing method level forward and backward slices, which are combined into complete slices using the Merge tool. To compute the different DFC relations, first the DCall tool with HotSpot virtual machine is used to generate the method level trace. This trace is processed by the JImpact tool that implements our global DFC algorithm, generating the two basic relations. The tool called Combine then generates 7 other relations including the final DFC (the others are not presented in this article). Finally, using the corresponding method level slices as actual couplings precision, recall and other values and diagrams are produced. Naturally, we used only forward slices when investigating EA, and complete slices for ER relations. At the points where "UNION" appears in the picture the combined data is produced for multiple test cases, as outlined in Section 3.

5.2 Precision and Recall

Precision measures what part of the impact sets computed by the algorithm is covered by the actual couplings. In other words, it shows the rate of true positives in the resulting impact sets, which may contain false positives as well. On the other hand, recall measures what part of the actual couplings are covered by the computed impact set, namely the rate of true positives over the total amount of actual dependencies (this may include false negatives). There is a trade-off between these two measures and our parameter d provides the way to set the desired type of efficiency.

We measured the precision and recall of all possible combinations of the two base matrices, but only three combinations turned to be useful: $EA_{call} \cup EA_{ret}$ to verify the efficiency of this simplified version of the method with call-indirections only, EA as the original Execute After algorithm and ER for DFC itself. Although computing only call-indirections would be useful because of small computational cost, it captures only a small part (22–28% at $d = \infty$) of the dependencies.

Figure 5 shows the overall results for ER for the test pro-

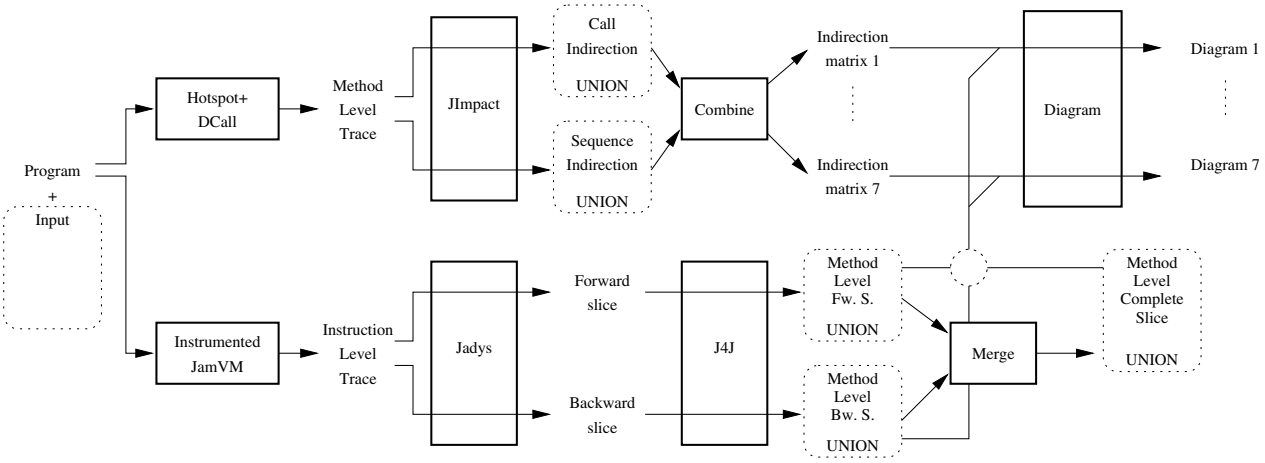


Figure 4. Measurement tools

grams. The other relation types showed similar shapes of the curves but with different values. It is interesting to conclude also that the three programs produced very similar results, although with different key values. It can be observed that as d grows recall also steadily grows with a relatively long linear phase, and at some point before ∞ (which is different for the different programs) it reaches 100%. This suggests that there is a threshold level of d for every program with which our algorithm can be used with safety.

On the other hand, precision starts at a higher value and rapidly decreases towards the precision of the original Execute After algorithm (but remaining somewhere above it because of EB relations), having a relatively long constant period. Our initial assumption before performing the experiments was that recall will reach a higher value before precision starts to decline, however the opposite happened. So we draw slightly different conclusions from the experiments, as summarized in the next section.

5.3 Impact set size

Naturally, precision and recall are important, but what really matters when it comes to applying this method is the sizes of the impact sets produced, since this will determine the efficiency of the software engineering task in question. We computed the set sizes for all d levels, however since the tendencies of their change were very similar to the recall curves, we will present only the interesting values ($d = 1, 2$) for ER in a table below. The mentioned similarity can be attributed to an interesting relation between impact set sizes, precision and recall. Namely, $impact = \frac{recall}{precision} \cdot C$, where C is a constant value. Since it can be observed from the precision curves that after a few steps of increasing d it becomes nearly constant, it follows that the impact set size will grow proportionally to recall in that period, and that it will eventually reach a certain value when recall reaches 100% (after this its precision will surely decline by moving towards ∞). Unfortunately, the sizes of the impact sets at 100% recall are not significantly smaller than that produced by the original conservative method.

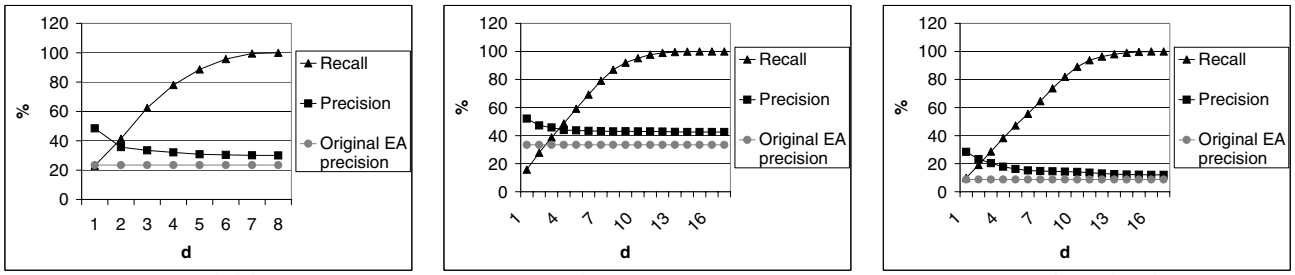
	$d = 1$			$d = 2$		
	set size	orig. p.	prec.	set size	orig. p.	prec.
JSubtitles	14.3	23.4	48.4	34.5	23.4	35.6
NanoXML	13.5	33.4	52.1	25.5	33.4	47.2
java2html	4.2	8.7	28.5	9.9	8.7	23.2

Table 1. Impact set sizes. The set sizes are shown as percentage values relative to the respective set sizes of the conservative method, whose precision values are shown in the second and fifth column. The third and the sixth column are the respective precisions for ER.

In Table 1 the average sizes and precisions over all functions' impact sets are shown. It can be seen that with $d = 1$ the precision of ER is at least twice better than that of the original Execute After method (with java2html much more), and that it is still very good with $d = 2$ as well. At the same time, the corresponding impact set sizes are much smaller, although the recall is not very good compared to the 100% of the original Execute After. However, this suggests that using $ER^{(d)}$ relations is a much better alternative to simply reducing the original Execute After sets by chance.

6 Discussion

Before we obtained our experimental results we had different expectations than what we actually realized by the end of the work. Namely, we thought that by reaching a good enough recall with a given d value, the precision will not start to decline significantly. Our results show that the



(a) JSubtitles (b) NanoXML (c) java2html

Figure 5. Precision and recall for ER (%)

opposite happened: precision declines very fast, while recall steadily rises for some more indirection levels. The outcome of this is that one should not aim at very high recall values using this approach, since the gain in terms of impact set size and precision is minor with respect to the safe and simpler Execute After method. However, with small d values (1, eventually 2), the gain is notable: the average impact set size is much smaller than the one with the safe method, and compared to using the safe method's impact sets that are arbitrarily reduced to the same size, double precision can be obtained with our method.

To summarize our findings, here are the answers to the research questions set up at the beginning of this article:

RQ1 Yes, a small DFC value between two functions indicates that there may be an actual coupling between them with higher probability. According to our measurements, all actual couplings can be identified by the level of maximum 5–15. The recall rates we obtained show that, except for very close indirection 1–2, by extending the cut-off value, higher levels uniformly bring new dependencies in with approximately the same precision. In other words, most notably DFC levels 1 or 2 indicate significantly more actual couplings than higher levels.

RQ2 If we observe higher indirection levels, the call-indirections alone do not represent many of the actual couplings (at the widest cut-off only about 20% is recalled). This means that a significant part of actual couplings comes from sequence-indirections, so the more complex algorithm incorporating all kinds of indirections is required, and the simple only-call algorithm is not sufficient. Still, if the constant complexity of the latter is preferable over the logarithmic complexity of the former, it may have its useful applications.

RQ3 The cut-off value of parameter d around 5–15 produces recall near 100%, having a similar precision at this point to the safe method (20–30%). However, this precision is reached much earlier, and remains approximately the same.

RQ4 The impact set sizes increase in a similar rate to the recall, namely a small d will produce small sets with proportionally smaller recall as well. The best precision values can be obtained at levels one or two.

RQ5 In terms of the size of the impact sets, the relative gain compared to the conservative approach is also scalable with a characteristic similar to the recall values. Namely, the closest level 1 produces impact sets that are on average 13–15% of the set sizes of the safe method, while level 2 brings in about 25–35%.

7 Conclusion and future work

In this paper we propose a parameterized measure of dynamic function coupling (DFC), which can be used to compute forward and backward impact sets on function level based on execution traces. The method has an adjustable precision and recall rate, which enables its use in different application scenarios. To verify the validity of the approach we performed a number of experiments, whose main aim was the investigation of the quality and size of the impact sets computed. The answers that we provide to the research questions underline the metric's usefulness in various fields related to software evolution of large software systems.

However, the basic purpose of this paper was merely to introduce the concept of the DFC metric. Accordingly, there are many open issues that need to be addressed in the future. The discussion of the method in this paper imposed a constraint on the layout of function call events in the traces and assumed single threaded programs. We feel that these constraints do not affect the validity of the approach for unconstrained real-life execution traces, they merely simplified the description. In the current implementation we have already tackled several problems related to these constraints, and we have also elaborated an extension of the method to multi threaded programs. In the future we plan

to investigate in more detail any impacts of these issues on the validity of the method.

In order to be applicable in real life usage scenarios, the scalability of the method also needs to be verified. In case studies on large software and different applications related to software evolution outlined in this article, we plan to measure performance attributes of the different variants of the algorithm, in addition to our current theoretical complexity findings. This way we could also gain some more insights into the method's applicability and efficiency in different application scenarios compared to other existing methods. Furthermore, more investigation is needed on the sensitivity of the approach to different attributes of the test cases such as their coverage.

The paper dealt with only dynamic analysis, however we feel that it would be useful to compare these methods to static approaches as well. Finally, we started to work on the static counterpart of Execute Round relation and DFC metric, which will eventually provide a more complete view on the topic.

Acknowledgements

This work was supported, in part, by national grants NKFP-2004 2/013/04 (CREG++), GVOP-3.1.1.-2004-05-0345/3.0 (OpenOffice++) and grant no. RET-07/2005 of the Péter Pázmány Program of the Hungarian National Office of Research and Technology.

References

[1] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. A. London. Incremental regression testing. In *Proceedings of the IEEE Conference on Software Maintenance*, Montreal, Canada, Sept. 1993.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 432–441, May 2005.

[3] E. Arisholm, L. C. Briand, and A. Føyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30(8):491–506, 2004.

[4] Á. Beszédes, T. Gergely, and T. Gyimóthy. Graph-less dynamic dependence-based dynamic slicing algorithms. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'06)*, pages 21–30, Sept. 2006.

[5] D. Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8):498–516, 1997.

[6] S. A. Bohner and R. S. Arnold, editors. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[7] B. Breech, A. Danalis, S. A. Shindo, and L. L. Pollock. Online impact analysis via dynamic compilation technology. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM'04)*, pages 453–457, Sept. 2004.

[8] L. C. Briand, J. W. Daly, and J. K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[9] L. C. Briand, J. Wüst, and H. Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings of the International Conference on Software Maintenance (ICSM'99)*, pages 475–482, Sept. 1999.

[10] J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich. JRipples: A tool for program comprehension during incremental change. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05)*, pages 149–152, May 2005.

[11] R. Gupta, M. J. Harrold, and M. L. Soffa. An approach to regression testing using slicing. In *Proceedings of the Conference on Software Maintenance 1992*, pages 299–308, 1992.

[12] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, pages 308–318, May 2003.

[13] Á. Mitchell and J. F. Power. A study of the influence of coverage on the relationship between static and dynamic coupling metrics. *Science of Computer Programming*, 59(1-2):4–25, 2006.

[14] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering held jointly with 9th European Software Engineering Conference (ESEC/FSE'03)*, pages 128–137, Sept. 2003.

[15] V. Rajlich and P. Gosavi. Incremental change in object-oriented programming. *IEEE Software*, 21(4):62–69, 2004.

[16] X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'04)*, pages 432–448, Oct. 2004.

[17] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[18] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, 1997.

[19] A. Szegedi and T. Gyimóthy. Dynamic slicing of Java bytecode programs. In *Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, pages 35–44. IEEE Computer Society, Sept. 2005.

[20] L. White and K. Abdullah. A firewall approach for the regression testing of object-oriented software. In *10th International Software Quality Week (QW'97)*, page 27, May 1997.

[21] L. White, K. Jaber, and B. Robinson. Utilization of extended firewall for object-oriented regression testing. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 695–698, Sept. 2005.